AFRL-IF-RS-TR-2003-269
Final Technical Report
November 2003

JOINT BATTLESPACE INFOSPHERE REPOSITORY PROTOTYPE

Northrop Grumman Information Technology

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 074-0188*

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** NOVEMBER 2003	**3. REPORT TYPE AND DATES COVERED** Final Mar 02 – Aug 03

4. TITLE AND SUBTITLE JOINT BATTLESPACE INFOSPHERE REPOSITORY PROTOTYPE	**5. FUNDING NUMBERS** C - F30602-00-D-0159/TASK 8 PE - 62702F PR - JBIT TA - PR WU - 08
6. AUTHOR(S) Patrick K. McCabe, Douglas J. Barnum, and Robert Gann	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Northrop Grumman Information Technology Defense Mission Systems 12005 Sunrise Valley Drive Reston Virginia 20191-3404	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/IFEA 26 Electronic Parkway Rome New York 13441-4514	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER** AFRL-IF-RS-TR-2003-269

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Patrick K. McCabe/IFEA/(315) 330-3197/ Patrick.McCabe@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.	**12b. DISTRIBUTION CODE**

13. ABSTRACT *(Maximum 200 Words)*

The purpose of this task was to design and develop a prototype capability that provides repository services for a Joint Battlespace Infosphere (JBI) Repository Prototype. Repository services include storage and retrieval of Information Objects as well as all data required for JBI operation, maintenance, and management. The objective in building the prototype was to obtain an understanding of the requirements for a robust JBI repository, and to examine in a laboratory setting, how those requirements could be satisfied.

The approach was to leverage in-house research and development for repository design and core service definition as points of departure for a far term design. Several repository software packages were evaluated to aid in the definition of the design space for the JBI repository prototype.

14. SUBJECT TERMS Joint Battlespace Infosphere, Repository, Core Services, Publish, Subscribe, Query, Control	**15. NUMBER OF PAGES** 30
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	**18. SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	**19. SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	**20. LIMITATION OF ABSTRACT** UL

Table of Contents

Table of Figures

Introduction

The purpose of this task was to design and develop a prototype capability that provides repository services for a JBI. Repository services include storage and retrieval of Information Objects[1] as well as all data required for JBI operation, maintenance, and management. The objective in building the prototype was to obtain an understanding of the requirements for a robust JBI repository, and to examine in a laboratory setting, how those requirements could be satisfied.

The approach was to leverage in-house research and development for repository design and core service definition as points of departure for a far term design. Several repository software packages were evaluated to aid in the definition of the design space for the JBI repository prototype. These packages include:

- Berkeley DB XML
- Xindice
- eXist
- PostgreSQL
- JavaCC
- An XPath Parser Grammar
- Castor
- Repository In a Box (Rib)

The Berkeley DB XML[2] package is a data store for XML documents. Xindice (pronounced zeen-dee-chay) is also a XML data store supporting the XML:DB interface API[3]. PostgreSQL is an open source relational database[4]. Java compiler-compiler (JavaCC) is the most popular parser generator for use with Java applications[5]. A free XPath parser grammar for the JavaCC tool is available and used for our research[6]. Castor is a library that allows a binding between Java objects and XML documents[7]. The Repository In A Box (RIB) is a repository mostly used for web-based sharing of software components and libraries[8]. More details about all of these technologies and how they were used later.

The JBI in-house team developed and released a JBI Common Application Program Interface (CAPI) in support of the Organically Assured and Survivable Information Systems[9] (OASIS)

[1] See the "Mercury Capability Guidelines," 22 January 2003, chapter 3 "JBI Information Object Model – Mercury Class" and chapter 6 "Persistence Management"
[2] See http://www.sleepycat.com for more information.
[3] See http://www.xmldb.org and http://xml.apache.org/xindice for more information.
[4] See http://www.postgresql.org for more information.
[5] See http://javacc.dev.java.net for more information.
[6] See http://www.fatdog.com for more information.
[7] See http://castor.exolab.org for more information
[8] See http://www.nhse.org/RIB/index.html for more information.
[9] See http://www.darpa.mil/ipto/programs/oasis_demval/index.htm for more information.

Demonstration and Validation project, referred to as version 1.0 of the common API (JBI CAPI v 1.0). JBI CAPI v 1.0 provides a publish, subscribe, query based infrastructure for application interaction in the OASIS demonstration and validation testbed. CAPI v 1.0 is an ideal foundation on which to build the JBI repository prototype, providing a complete platform that allows rapid insertion of new repository techniques and software components with low to moderate technical effort.

Repository Overview

The repository for the Joint Battlespace Infosphere (JBI) platform provides the working memory for all JBI core services and is especially critical for the query core service. For the query service to work at all, all information objects or suitably complete descriptions for all published information objects must be available for interrogation. The repository therefore, must contain those information objects (or their descriptions, see Figure 1 "JBI Repository Role"). Critical attributes of any JBI repository implementation include scalability to thousands of information object types, millions of information objects, and resistance to network and system performance fluctuations. Additionally, the repository must support Information Object archival, access and utilization auditing for security purposes, and sharing of information for performance optimization purposes.

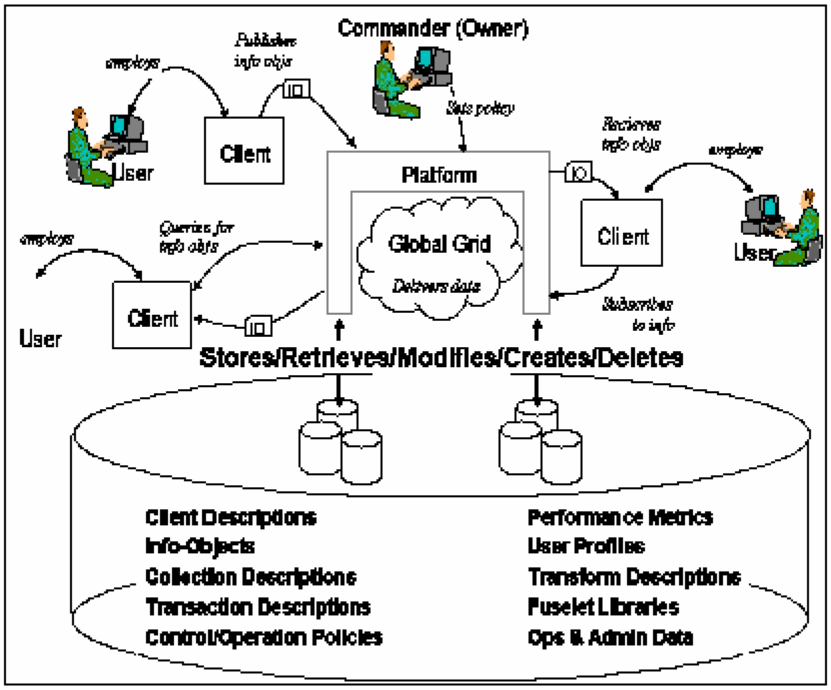


Figure 1 - JBI Repository Role

As shown in Figure 2, "Relationship of Application and Repository Interfaces", the approach was to make a generic, albeit robust interface to the repository available to the core service implementation. This approach has a number of advantages; it is relatively immune to variations in core service implementations, tolerant of changes to the Common API, and repository technology neutral.

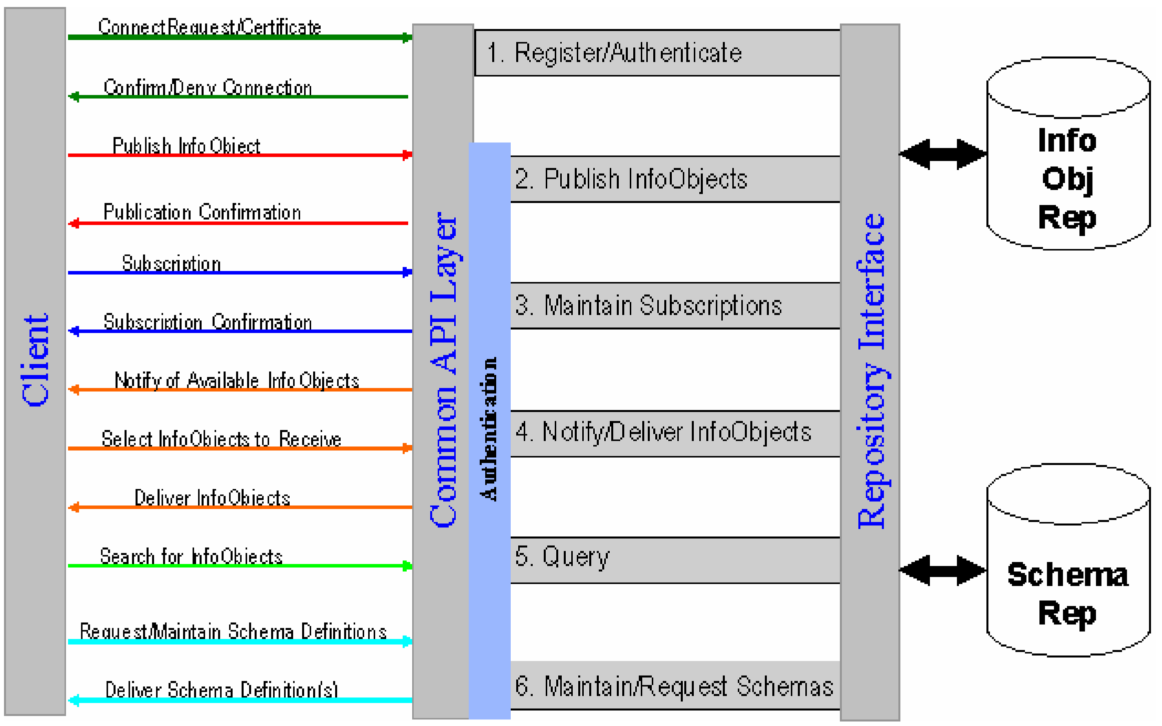

Client
Connect Request/Certificate
Confirm/Deny Connection
Publish Info Object
Publication Confirmation
Subscription
Subscription Confirmation
Notify of Available Info Objects
Select InfoObjects to Receive
Deliver InfoObjects
Search for InfoObjects
Request/Maintain Schema Definitions
Deliver Schema Definition(s)
Common API Layer
Authentication
1. Register/Authenticate
2. Publish InfoObjects
3. Maintain Subscriptions
4. Notify/Deliver InfoObjects
5. Query
6. Maintain/Request Schemas
Repository Interface
Info Obj Rep
Schema Rep

Figure 2 - Relationship of Application and Repository Interfaces

Archiving Information Objects should be a simple and efficient process. A substantial amount of resources have been spent, and are currently dedicated to efficient storage and manipulation of data[10].. Any JBI repository prototype should leverage this huge commitment of resources and existing infrastructure. Within the JBI, data persistence should be a platform service, not an application responsibility. This ensures consistent, platform wide services for all applications and allows the exploitation of the performance and quality of any number of third party commercial products

The focus of the repository prototype effort is a flexible and robust interface that is portable and allows for alternative repository implementations that provide creative and efficient Information Object storage, retrieval and sharing. Current data stores offer different paradigms or strategies to perform the storage management function. These include relational, object-oriented and XML document structured solutions. A JBI repository design must not tie itself to any one data store paradigm at this point in time. This approach allows for experimentation with a variety of data store and storage management solutions. These solutions are implementable with a minimum level of effort since MOST of the heavy lifting is abstracted away from application and platform by these readily available storage management solutions. The repository implementation is the

[10] The worldwide market for DBMS software is on the order of $13B per year "New fronts open in database war" By Ed Scannell and Tom Sullivan, InfoWorld, October 20, 2000 1:01 pm PT Clearly, this figure does not include the enormous resources devoted to the implementation, operation, and maintenance of the repositories built with these products.

"glue" between these data store systems and the JBI proper.

The repository itself cannot live in a vacuum. It must be developed and tested in a real JBI implementation. There have been two JBI implementations built in-house. The first was the Jini Pub/Sub and the second is the OASIS 1.0 release. The Jini Pub/Sub implementation was developed to better define the characteristics of publish and subscribe core services in a JBI context. Principle areas of investigation where concepts of metadata invariance and the characteristics of the broker function in a publish/subscribe service. In this implementation, there was no repository per se – JMS persistence was utilized, but this function is oriented towards insuring object delivery, not to support a comprehensive query service[11].

The OASIS implementation builds on the understanding gained developing the Jini Pub/Sub implementation. This implementation however was built using J2EE technologies and a full blown repository interface based on Oracle 9i. Developed as part of this implementation were an information object repository and associated information object schema repository (referred to as the metadata repository by the development team and their extended development family). The repository implementation is Oracle specific however, so additional research is oriented towards a more generic repository interface[12]. The OASIS release is cutting edge and an excellent foundation for our repository work. The current approach is to replace the OASIS repository code and objects with code and objects that abstract details of the repository implementation away from the common API. This approach enables the development and testing of new repository solutions.

Information Object Repository Interface and Driver API

The end goal of a Repository application (client) interface, a robust and full functioned exploitation of the JBI Repository, is a very large undertaking. However, to be able to implement some capability that would be easily tested, a Repository interface was designed that closely resembled the OASIS 1.0 Information Object Repository (IOR) facade interface. This would allow software to be created that could easily plug into OASIS and enable the testing of solutions other than the solution selected by the OASIS team for the Information Object Repository.

The Information Object Repository Interface is very simple and allows both archiving and querying capabilities to a user. This functionality is enabled by the following methods:

```
boolean archive(String type, String version, String metadata, Object object);
ResultSet query(String type, String version, String query);
ResultSet query(String queryID, int pageSize, String type, String version, String query);
```

[11] "A Jini-Based Publish and Subscribe Capability" Vaughn T. Combs and Dr. Mark Linderman

[12] "JBI OASIS Version 1.0 for the Organically Assured and Survivable Information Systems (OASIS) Demonstration and Validation (Dem/Val), Distribution CD #1: Platform Services," 31 Mar 2003, AFRL/IFSE, JBI Program Office

To archive an Information Object, simply supply four pieces of information; the type of Information Object, the version of the type of Information Object, the metadata describing the Information Object and finally the actual Information Object.

There are two query methods available. The first is a simple query that supplies three pieces of information. The type of Information Object to query, the version of the type of Information Object and finally a query String to apply to the Information Objects contained in the Repository. All the known Information Objects contained in the Repository are returned in a ResultSet object. The second query method allows the user to "page" through the query ResultSet which improves performance and memory requirements.

At this time, the query language used by OASIS is XPath. The query language implementation is OASIS specific, hence the current implementation of a query language in the repository prototype requires XPath. The JBI repository prototype design however, does not require the specificity of the OASIS implementation, only that the query be representable as a String. The String is simply passed to the lower level data store where processing the query can be abstracted away from the application.

The atomic unit of information objects stored in the repository prototype is the base java.lang.Object instance, which every Java object must extend. An Information Object therefore, is not limited to any particular object definition. Persistence of information objects are handled at the lower levels of the data store implementation. The only requirement for an Information Object in the JBI repository prototype is that Information Object must implement the java.io.Serializable interface. This ensures that lower level data stores will simply serialize an Information Object instance so it can be persisted. OASIS is built upon Java 2 Enterprise Edition (J2EE) technologies therefore all objects passed among J2EE entities over the network MUST implement java.io.Serialization by definition, so the implementation is not constrained. Declaring the argument as a java.io.Serilizable is not a strict limitation in the JBI repository prototype interface since it is desirable for the interface to be valid in a non-J2EE environment.[13]

The Information Object Repository Interface also can supply some information about the Repository to a JBI client, such as the number of stored Information Objects by type, and enumerations of types, information object versions, and usage statistics. Clients can obtain basic information about the state of the Repository, and hooks are provided for a direct information management staff interface in support of the control core service. This functionality is enabled by the following methods:

long getInfoObjectCount(); Returns the total number of information objects contained within the repository.
long getInfoObjectCount(String type, String version); Returns the number of information objects contained within the repository that have the specified type and version.
InfoObjectTypeVersion() getInfoObjectTypeVersions(); Returns an array containing all known information object types and associated versions contained in the repository.

[13] In an embedded spaces environment for example, such as t-spaces, java spaces, and Linda-Spaces

The Information Object Repository Interface is just that, an interface[14]. No code is really implemented, the interface simply defines the methods that need to be provided by an implementing class to fulfill the requirements of an Information Object Repository. The package contains a class called Driver which implements some of the common functionality ALL implementations of the interface must provide. Each implementation is expected to extend Driver and implement the remaining functionality. The Driver class is an abstract Java class that supplies some common basic plumbing methods and functions. The extensions need to supply the method implementations of the Information Object Repository Interface since only the extensions truly know what to do to interact with their specific data store. By designing the infrastructure in this way, a plug in type of architecture is allowed. Many extensions to the Driver class can be implemented, limited only by the imagination of the developer. If a data store exists, a Driver could be written to use it. Imagine a Driver that could access any data store supplying a JDBC interface. Or a Driver that itself is a data store by maintaining data in flat file of the given operating system. These extensions can then perform their required functionality in different ways. Different data store implementations could be provided and abstracted away from our Information Object Repository Interface design. In a runtime setting, these different implementations could be compared for performance and resource utilization rather easily using the same hardware, plug in some implementation, then measure its performance using a common criteria[15].

There were two implementations done in this task. They used two different underlying data store technologies which were the basic differences in the implementations. The data stores were the Berkeley DB XML and the XML:DB implementation Xindice from the Apache open source group. A third data store was evaluated called eXist, another XML:DB implementation but had limitations and was unusable.

Berkeley DB XML

Sleepycat Software[16], is the maker of Berkeley DB, which is one of the leading embedded data management software in the world. They recently released version 1.0 of Berkeley DB XML, a high-performance extremely reliable embedded database engine that stores and manages XML data.

Berkeley DB XML is a library that links directly into the user's application, thus providing superior performance by eliminating communications among processes or systems.

Documents are stored in collections. A collection is a set of XML documents. There is no other requirement than to simply input N number of XML documents to the collection. The documents can be totally disjoint, meaning they do not need to conform to the same schema.

[14] Essentially, a class that provides operations without methods, utilized by external classes in order to exercise some service such as JDBC.

[15] There are well defined repository benchmarks and an excellent source of information can be found in "The Benchmark Handbook", http://www.benchmarkresources.com/handbook/introduction.asp. Unfortunately, these benchmarks apply to well defined problems not directly applicable to the JBI Repository.

[16] See http://www.sleepycat.com/ for more information.

However, it would be more efficient and a better design of the XML database if the documents were related or conformed to the same schema. The DB XML database is used efficiently by having a collection for each type and version of the defined Information Objects. A single application can operate on many collections at the same time. This is an important feature because there is a need to maintain many collections since there will be many types and versions of Information Objects.

Non-XML data may be included by creating standard Berkeley DB tables. Tables and collections may be used together, with full support for transactions and recovery services by multiple users simultaneously. There is a need to store the information object payload, and payload can be stored in the standard tables provided by Berkeley DB, it not required that the payload itself be an element of the XML document.

Berkeley DB XML's Query Processor implements XPath 1.0[17]. A cost-based query optimizer considers the indices that exist, the data volume that a query is likely to produce and the cost of computation and disk I/O to select a query plan with the lowest run-time cost. As a product that would be considered for a Repository instance, this careful design and implementation of this algorithm by Sleepycat Software is something desirable to leverage.

Other key features:

- Supports Windows NT/2000/XP, Linux and Solaris.
- Includes complete source code.
- Manages XML documents quickly and reliably with a high-performance embedded data manager.
- Stores and retrieves native XML documents. No conversion to relational or object-oriented models required.
- Combines XML and non-XML data in a single database.
- Supports XPath 1.0 and other W3C standards for XML and XML Namespaces.
- Provides C++ and Java APIs.
- Supports multiple threads per process, and multiple processes per application using a thread-safe library.

Berkeley DB and Berkeley DB XML packages are considered Open Source software and can be acquired for free. An informal mailing list exists for simple support questions and feedback from a community of users who have adopted the use of these packages. To use the Berkeley DB XML as a data store, an extension to the Driver abstract class of the Information Object Repository package was implemented. It was called the DBXML driver.

The DBXML driver that was developed allowed the use of Berkeley DB XML as a Repository instance. The DBXML driver used the Berkeley DB XML data store to maintain the Information Object metadata. The whole Information Object was serialized to a byte array and stored as binary data in a Berkeley DB table.

[17] See http://www.w3.org/TR/xpath for the complete specification.

Each instance of an Information Object type and version constituted a Berkeley DB XML collection, and an associated Berkeley DB table was implemented to hold the actual instances of Information Objects.

Persisting a new Information Object instance results in its metadata being placed into a Collection determined by it's type name and version value. Upon successful insertion, the Berkeley DB XML package assigns the metadata a unique ID. This ID value is used as the primary key for the serialized Information Objects stored in the appropriate Berkeley DB table. This allows for easy retrieval of the actual Information Object instances upon completion of a successful query at some future time.

The Information Object instance being archived is placed into a DB XML Collection by type and version. If given an Information Object instance has a type and version not currently known, a new Collection is created on the fly to hold this new Information Object.

The Berkeley DB XML package supports XPath natively. It was a simple matter to pass the user supplied XPath query string on invocation of the query methods to find the ResultSet of Information Object instances that satisfy the XPath query. No conversion or parsing of the XPath expression was needed to be done by the driver code to process the query.

Xindice

Xindice from the Apache project[18] is a database designed from the ground up to store XML data, and is an example of what is more commonly referred to as a native XML database. The benefit of a native solution is that you don't have to worry about mapping your XML to some other data structure, such as relational or object-oriented data stores. The data is inserted as XML and retrieved as XML.

Xindice also uses XPath (http://www.w3c.org/TR/xpath) for it's query language and XML:DB XUpdate (http://www.xmldb.org/xupdate/index.html) for its update language. An XML:DB API implementation is provided for Java development. The XML:DB API is an open API designed by the XML database industry. Since XML databases represent a new technology there has been up to this point, no concerted effort to develop specifications targeted for the XML market. The lack of specifications inevitably increases the learning curve for employees, prevents product interoperability and ultimately slows the adoption of the products in the market place. To address these issues, a decision was made to start the XML:DB initiative (http://www.xmldb.org). It is hoped that through the efforts of XML:DB, that standards can be developed for the XML database industry and that XML databases can make it into the standard tool set used by IT departments worldwide.

An XMLDB driver was implemented so Xindice and other XML:DB data stores could be used as Repositories. The XMLDB driver is instantiated with properties that allow it to interact with any XML:DB type of data store including Xindice. This would mean that we could have every

[18] See http://xml.apache.org/xindice/ for more information.

XML:DB data store be used as a Repository. It was found that because the XML:DB API interface is immature, this generic XMLDB driver is not really possible at this time. The XML:DB API interface needs more capability before this can truly happen. For example, there is no standard way to add new Collections on the fly at run-time. The XMLDB driver needed to access core Xindice API objects and methods to enable this capability since the XML:DB definition did not allow for this capability. Surely this is an over site on the XML:DB committees part since this is a basic and fundamental task. Because of this limitation, the XMLDB driver at this time only worked with Xindice.

The XML:DB specification also allows for the mixture of both XML data and binary data. The API defines this, but it was found not to be implemented very often by the XML:DB data stores that were investigated. Xindice does not support the functionality of storing binary files. Since the payload would generally be stored as a binary file, this functionality is required. The XMLDB driver has code to support this functionality with payload stored on the local file system. This solution is less than ideal, since the burden of maintaining these files falls upon the XMLDB driver code, which was not developed by a third party and must be supported by us or someone else. It would be much more desirable for this functionality to be supplied by the XML:DB vendor so the improvements in their releases could be taken advantage of as available. Until then, the driver must take this responsibility. In the future, it is expected that XML:DB data stores will support this part of the API and the XMLDB driver will not be burdened with this responsibility.

eXist

eXist 0.8.1 is a Java-based, open source native XML database that is suited for XML collections that are occasionally updated. eXist has pluggable storage back ends, index-based XPath query processing with extensions provided to support keyword search. Several interfaces come with the database including HTTP, XML-RPC, SOAP and WebDAV. Two implementations of the XML:DB API have been provided. The first talks to a remote database engine using XML-RPC calls. The second has direct access to a locally running database instance. This option would allow a developer to embed eXist into a stand-alone application without running an external process.

eXist has been designed to be a pure native XML database, even though it provides an optional relational storage backend. Database broker classes handle all calls to the storage back ends (reference figure). These classes provide a limited set of basic operations. In addition, methods are provided to access available index structures.

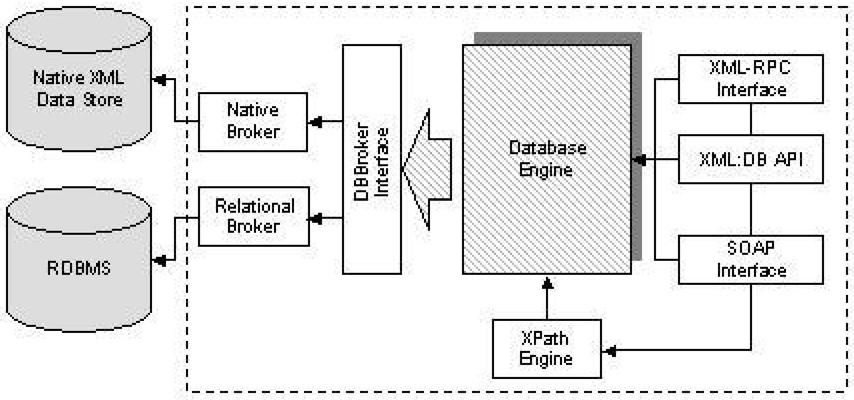
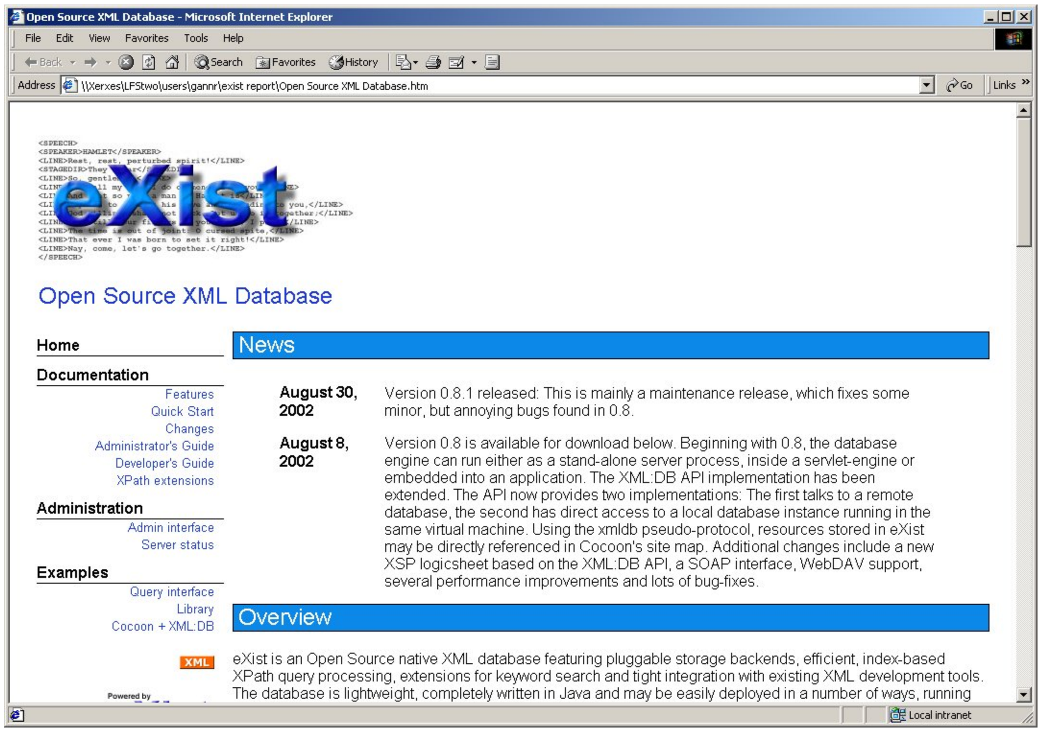


Figure 3 – eXist Architecture

Installation was straightforward. To install, Java 2 is required (can be found at http://java.sun.som/j2se/1.4.2/download.html). eXist has been tested with several operating systems including Solaris 8, Windows 2000, and Linux. It has also been tested with both Sun's and IBM's JDK. It was installed on jbix1, a Compaq server running Windows 2000 Advanced Server. eXist can be run in three different ways: as a stand-alone server process, within a servlet engine or directly embedded into an application. In task 8 it was run as a stand-alone server process. By making this choice, installation was as easy as unzipping the archive and setting two environment variables: JAVA_HOME and EXIST_HOME. The JAVA_HOME environment variable is set to the directory where either the Java Runtime Environment (JRE) is installed. The EXIST_HOME is set to the directory where eXist is installed. To start the service, we executed the startup.bat script located in %EXIST_HOME%\bin. After executing this command, we verified the service was running using a web browser and the following URL:

http://localhost:8080/exist/index.xml

The following page was shown indicating success.


Open Source XML Database - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address \\Xerxes\LFStwo\users\gannr\exist report\Open Source XML Database.htm
Go Links
Local intranet

Open Source XML Database

Home

Documentation
Features
Quick Start
Changes
Administrator's Guide
Developer's Guide
XPath extensions

Administration
Admin interface
Server status

Examples
Query interface
Library
Cocoon + XML:DB

News

August 30, 2002	Version 0.8.1 released: This is mainly a maintenance release, which fixes some minor, but annoying bugs found in 0.8.
August 8, 2002	Version 0.8 is available for download below. Beginning with 0.8, the database engine can run either as a stand-alone server process, inside a servlet-engine or embedded into an application. The XML:DB API implementation has been extended. The API now provides two implementations: The first talks to a remote database, the second has direct access to a local database instance running in the same virtual machine. Using the xmldb pseudo-protocol, resources stored in eXist may be directly referenced in Cocoon's site map. Additional changes include a new XSP logicsheet based on the XML:DB API, a SOAP interface, WebDAV support, several performance improvements and lots of bug-fixes.

Overview

eXist is an Open Source native XML database featuring pluggable storage backends, efficient, index-based XPath query processing, extensions for keyword search and tight integration with existing XML development tools. The database is lightweight, completely written in Java and may be easily deployed in a number of ways, running

Figure 4 – eXist Start Page

Once the database was installed, a call was put out for XML files. A limited number of files were able to be secured that were stored on jbix1. Next, 3 collections were created in the database. A collection, similar to a Windows' folder, is simply a group of related XML files. The collections were titled CMAPI, GIS and USMTF. Then the collections were populated with the available data using the eXist command line client interface. XML files had to be generated from message transaction format files for the complete US Message Transaction Format (USMTF) collection[19]. There is a utility that follows the XML-MTF Mapping Public Working Draft dated March 2000 developed by the XML Development Team. After populating the collections, experimentation with the different user interfaces that were available were performed.

Executing the client.bat script found in % EXIST_HOME%\bin\ starts the command line client interface. This puts the user in a command window with a prompt. The functionality of this interface is very limited as only a few commands are provided. In this mode, the user can add and remove collections (mkcol, rmcol) and insert data – both a file at a time and a directory at a time. After using the interface for a brief time, the limitations became quickly apparent. First, the interface must be restarted to view the contents of a newly added collection. Second, the interface doesn't accept wildcards. Third, there is no *move* command. Whenever files are placed in the wrong location, they have to be deleted then added again using the put command. A fourth limitation is that the file names cannot have spaces in them since Windows interprets spaces as delimiters.

[19] See http://www-usmtf.itsi.disa.mil/ for additional information.

Going operational with this database would be relatively easy requiring just an Apache server with Tomcat to be set up. The database is very easy to use, and can be up and running in literally minutes.

At this point however, eXist cannot be recommended as a suitable operational capability. Even though it is very easy to use, the functionality seems to be very limited. This database is well suited as a working prototype but not much more.

Using OASIS 1.0

To test the software modules developed for this task, it was convenient to take advantage of the development of the OASIS 1.0 release by the in-house JBI team. A significant amount of setup effort can be avoided by using the bulk of the OASIS implementation and changing a small number of classes and configuration files.

The OASIS 1.0 release used Oracle 9i for all database requirements. There are actually three database requirements needed. They are:

- Storage of users, passwords, roles and permissions, the "Security Database"
- Persistence of Information Object instances, the "Information Object Repository" (IOR).
- Maintaining defined Information Object Types, their metadata schema and versions, the "Schema Repository". The schema repository is referred to in the "Mercury Capabilities Specification" as the "Metadata Repository" or MDR. This title is not really an accurate reflection of the MDR role in the JBI.

For development and testing purposes, PostgreSQL was adapted for the storage of user information. For the persistence of Information Object instances, two data stores were used, the previously described Berkeley DB XML and Xindice. The Information Object schema were restored in PostgreSQL.

Using PostgreSQL instead of Oracle 9i for OASIS 1.0 user interactions.

Oracle 9i takes a lot of resources to be able to use, both in cost and hardware. Using a free and open source database would make it simpler to get OASIS 1.0 running on the development hardware available. One such available open source relational database is called PostgreSQL[20].

The PostgreSQL Global Development Group is a community of companies and people co-operating to drive the development of PostgreSQL, one of the worlds most advanced Open Source database software.

The PostgreSQL software itself had its beginnings in 1986 inside the University of California at Berkeley. In 16 years it has evolved from a research prototype to a significant player in the global database management software market, leveraging a globally distributed development

[20] See http://www.postgresql.org/ for more information.

model, with central servers based in Canada.

PostgreSQL was easily available for use. It was a matter of modifying some OASIS application server configuration files and modifying several initialization SQL scripts that come with the OASIS 1.0 distribution. Even though the 'S' in 'SQL' means 'standard', it was discovered that the Oracle SQL syntax used was not fully supported by PostgreSQL syntax. This modification to the SQL scripts was done as well as possible to allow the use of PostgreSQL instead of Oracle 9i for the security/roles functionality of the OASIS application server.

One particular problem was not solved. The OASIS application server could not authorize users using password encryption. It is believed that a Java platform bug exists having to do with the encryption libraries for Linux.

As a work around, the OASIS application allowed for encryption of these passwords to be turned off. This was done in our testing setup. An additional step had to manually perform to correctly place the unencrypted password in the PostgreSQL database table used for user names and passwords. After this, users were correctly validated.

Persisting Information Object instances via EJB Deployment

To test the implemented DBXML and XMLDB drivers and their underlying data stores in the OASIS 1.0 release, three Enterprise Java Beans (EJB) needed to be created and deployed in the OASIS application server. These beans replaced the OASIS 1.0 EJBs that use Oracle 9i resources. The EJB framework makes it rather easy to deploy these replacement EJB objects so the DBXML and XMLDB drivers to be tested and used.

Information Object Repository OASIS EJB

The OASIS 1.0 release does its repository work in a J2EE environment. The following diagram details the OASIS 1.0 structure for the repository EJB deployment.

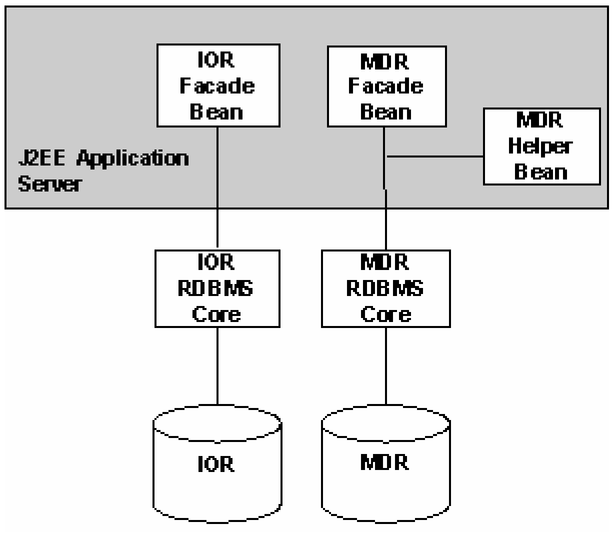


Figure 5 – OASIS High Level Architecture

The OASIS 1.0 release comes with an Information Object Repository (IOR) facade bean that when used can store Information Objects and their metadata to the Oracle 9i database. It is desired to replace this IOR facade bean with one of our own. This new bean is called RepositoryIORFacadeBean. When this deployable EJB is created, it is configured to load a particular driver that exercises the desired underlying data store. In this instance it has been configured to either use the DBXML or XMLDB driver. Again, this is a build time configuration. Meaning to change the actual driver, the RepositoryIORFacadeBean must be rebuilt and redeployed. For practical purposes, this is fine to do for testing purposes. But no Information Object instances are shared between the low-level data stores. For example, say that the RepositoryIORFacadeBean is deployed configured with the DBXML driver. Over S seconds of time, N Information Object instances are published and archived to the DBXML data store. The RepositoryIORFacadeBean is then redeployed but now configured to use the XMLDB driver. Those N number of Information Object instances would be "lost" and unknown to the currently running RepositoryIORFacadeBean. Any newly archived Information Object instances would now be stored in the XMLDB data store. Again, redeploying the RepositoryIORFacadeBean back configured with the DBXML driver would result in the original 5000 Information Object instances now being contained or accessible by the RepositoryIORFacadeBean that is currently running and any archived Information Object instanced published while the XMLDB was being used would now be "lost". A work around to this problem could be achieved by writing a simple program that could migrate the Information Object instances from one data store to another. The program would not need the J2EE infrastructure. It could do its work by interacting with the Driver API only. The migration could then be done offline or concurrently while the new data store is running.

Metadata Repository OASIS EJB

The OASIS 1.0 release comes with two MetaDataRepository (MDR) beans. One bean is a facade bean that is used by client objects. The other bean is a helper bean only used by other application server beans. In particular, the helper bean is used by the OASIS web application that allows users to control the state of the MDR. Functions include examining, updating and adding Information Object metadata definitions allowing for dynamic configuring and publication of new Information Object types and versions.

The standard OASIS 1.0 MDR beans used Oracle 9i as its default data store. Again it was desired to replace Oracle 9i as a requirement to executing OASIS 1.0. The MDR beans were replaced with RepositoryMDRFacadeBean and RepositoryMDRBean.

Even though the MDR and IOR repositories are different entities and have different functionalities, they both could be implemented by the same Information Object Repository Interface and Driver definitions previously discussed. The main difference between the two repositories is that the MDR is NOT required to support XPath as its query language. The basic functionality of the MDR is to store the JBI common API implementation SchemaObject. The SchemaObject is a simple container that has three properties. These properties are:

- The type of Information Object.
- The version of the type of Information Object.
- The actual metadata. The metadata is XML schema.

The querying requirements of the MDR, at least defined by the OASIS 1.0 release, is a rather simple query of the known SchemaObject instances contained in the MDR.

From earlier development, an object existed that is capable of taking any Java object and storing it in a JDBC style data store. This object is simply called Store. Given a particular Java object class, Store can create a database table schema that can maintain a column entry for each “property” of the given object. A “property” is defined in the Java object by that Java object supplying “getter/setter” methods. For example, say a particular Java object has a property called “Name” defined as a String type. The Java object MUST expose two public methods thusly:

```
public String getName();
public void setName(String s);
```

Every property in the given object that has methods in this way will be persisted in the JDBC database. The Store object using Java Reflection can determine a Java objects properties at run-time and can generate the proper SQL to create a table schema, an SQL insert statement to insert new object instances and execute the appropriate SQL select statements to query the database to find appropriate objects that satisfy the query.

The RepositoryMDRFacadeBean and RepositoryMDRBean each use an instance of the Store object to maintain a collection SchemaObject instances. These SchemaObject instances detail the currently defined Information Object types and versions known to the running OASIS 1.0 instance. OASIS supplies a web application that allow the viewing of the SchemaObject instances, their counts and their schema definitions. The web application also allows adding, removing and modifying of SchemaObject instances. This web application is in fact interacting with the RepositoryMDRBean instance running in OASIS.

These MDR beans have been configured to use two different JDBC databases. They are PostgreSQL and another pure Java open source database called HyperSQL[21]. It is expected that these beans would work with any JDBC style database, being a matter of a build time configuration.

The following diagram details the changes made in the OASIS 1.0 release to use these new EJB beans created.

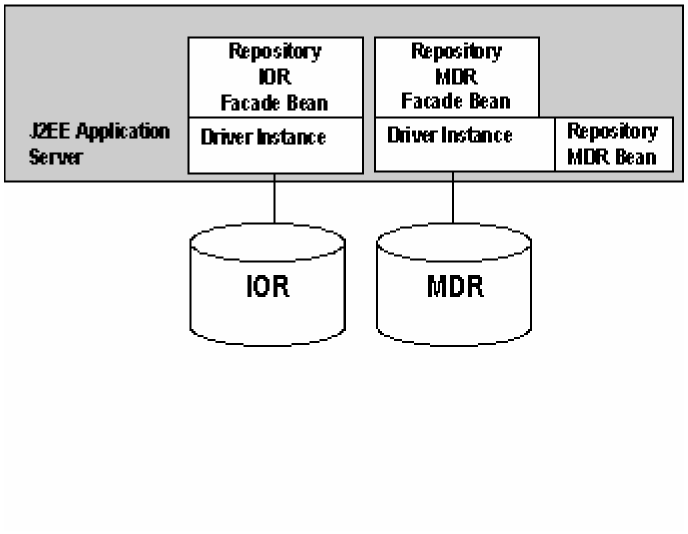


Figure 6 – OASIS 1.0 EJB Overview

XPath Parser Grammar and JavaCC

The two data stores we used had a very important common trait. They both support XPath querying of the metadata XML files maintained in their respective store. This is important because in OASIS 1.0, the default query language is XPath. Since the both data stores support XPath, this reduced our burden by eliminating the need to parse or process the input query. It was only necessary to pass the input query string to the data store and to return the resulting Information Object instances.

However, it would not be desirable to limit any data store that might be useful for our purposes. There are several very good object databases on the market that could be very efficient repositories for Information Object instances. Some commercial object databases would be ObjectStore (http://www.objectstore.net) and FastObjects (http://www.fastobjects.com/us). An open source object database called Ozone (http://www.ozone-db.org) might also be used. In most likelihood, these object databases would not support XPath since XPath is an XML standard.

To support data stores that do not offer XPath query capability, a module or plugin that converts

[21] See http://sourceforge.net/projects/hypersql/ for more information.

XPath into the query language the data store does support must be provided. An extra step, but something that might well be worth doing. An open source project called XQEngine from http://www.fatdog.com has developed an XPath parser grammar. The grammar is for JavaCC, an excellent parser generator tool originally from Sun, now an open source project itself and available at http://javacc.dev.java.net. Java compiler-compiler is the most popular parser generator for use with Java applications. A parser generator is a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar. In addition to the parser generator itself, JavaCC provides other standard capabilities related to parser generation such as tree building (via a tool called JJTree included with JavaCC), actions, debugging, etc.

The beauty of having such a grammar already created is that the task of parsing XPath to some other format is greatly reduced. By far the most difficult work to complete such a task is already done. An API framework could easily be imagined that would allow easy conversion of XPath to any form desired by the programmer. Using such an imagined framework, adding new data stores that did not support XPath natively would become a rather painless process and allow the use of efficient, future data stores.

Castor

A technology that was investigated in task 8 was an open source project called Castor. The Castor project is located at http://castor.exolab.org. Castor is an open source data binding framework for Java[tm]. It's basically the shortest path between Java objects, XML documents and SQL tables. Castor provides Java to XML binding, Java to SQL persistence, and then some more.

Castor was intriguing for its Java /XML binding. Since JBI Information Objects are described using XML and specified by an XML schema definition, the Castor product could allow seamless parsing of input XML metadata into Java objects. The Java objects could then be accessed via their methods to examine metadata values. As developers, no special parsing code would be required. Code to parse the XML would be generated by the Castor package. The Castor package only needed the XML schema definition to generate Java code that could parse given XML data conforming to the XML schema. These generated classes would be powerful performing all code conversions based upon the XML schema. If the XML schema defined a particular property as Boolean, then the property would be converted to a Boolean from a string doing parsing. It is very convenient to the programmer not to have to do these data type conversions on their own. This capability would also aid in debugging, given metadata that was not correct or did not conform to the defined schema.

Since both data stores we used were XML and XPath aware, Castor did not come into play with those solutions. However, if an object-oriented database was used, Castor would have been a huge aid in the repository implementation. Since all items stored in an object-oriented database are objects, the XML could be turned into objects via Castor then stored in the database. And recreating the original XML would also be a simple task because of Castor. A missing piece would be the querying capability. Something like the previous package mentioned created from the XPath parser grammar would need to be created to take XPath to the native object-oriented query language or method. These two packages would very much enable efficient use of object-oriented databases in the future with very little development effort.

The RIB

During task 8, a project called Repository In a Box (RIB) was investigated[22]. The RIB is a software package for creating web-enabled metadata repositories. Metadata is information that describes reusable objects, such as software packages or datasets. RIB allows the user to enter metadata into a user friendly java applet which then sends the information to a RIB server via HTTP. The information is then stored in an SQL database where it is automatically made available in a fully functional web site (catalog, search page, etc). Repositories which use similar data models can use the XML processing capabilities of RIB to share information via the Internet. Third party applications can access the data stored in RIB by using the RIB Application Programmer's Interface (API).

The RIB was created by a development team located at the University of Tennessee under direction by the National HPCC Software Exchange (NHSE). The RIB employs the Basic Interoperability Data Model (BIDM), IEEE Standard 1420.1. The purpose of BIDM is to define the minimal set of information about assets that reuse libraries should be able to exchange in order to support interoperability. Since the BIDM describes a minimal set, other data that would be useful for interoperability, such as library data model information, as well as communication protocol and related standards, are not included.

An instance of the RIB was installed on a development machine was determined that the querying capability of the RIB was not robust enough to use it as an actual Repository data store for Information Object instances. Since the RIB does not support XPath natively, a very inefficient scheme would have been required. Basically, the RIB query mechanism is something along the lines of "give me the contents of the RIB and the result set is returned as a particular XML document". An XPath expression could have been applied against this document however, as RIB contents expanded, it became very evident that query processing performance would suffer greatly.

The RIB showed its true capabilities when it was used to maintain the documentation and distribution of the source code developed for task 8. Using a RIB instance, it was possible to easily show others the classes, objects, associated documentation and usage. It was shown that a tool like the RIB would be a wonderful collaborative repository for software developed at RRS and would make a great campus wide resource. So much code locally is developed "in a vacuum", that a RIB would allow for substantial code sharing and reuse, increasing overall productivity for lab scientists and engineers.

The initial RIB data model detailed in the following diagram is an extension of IEEE 1420.1, the Basic Interoperability Data Model (BIDM).

[22] See http://www.nhse.org/RIB/index.html for more information.

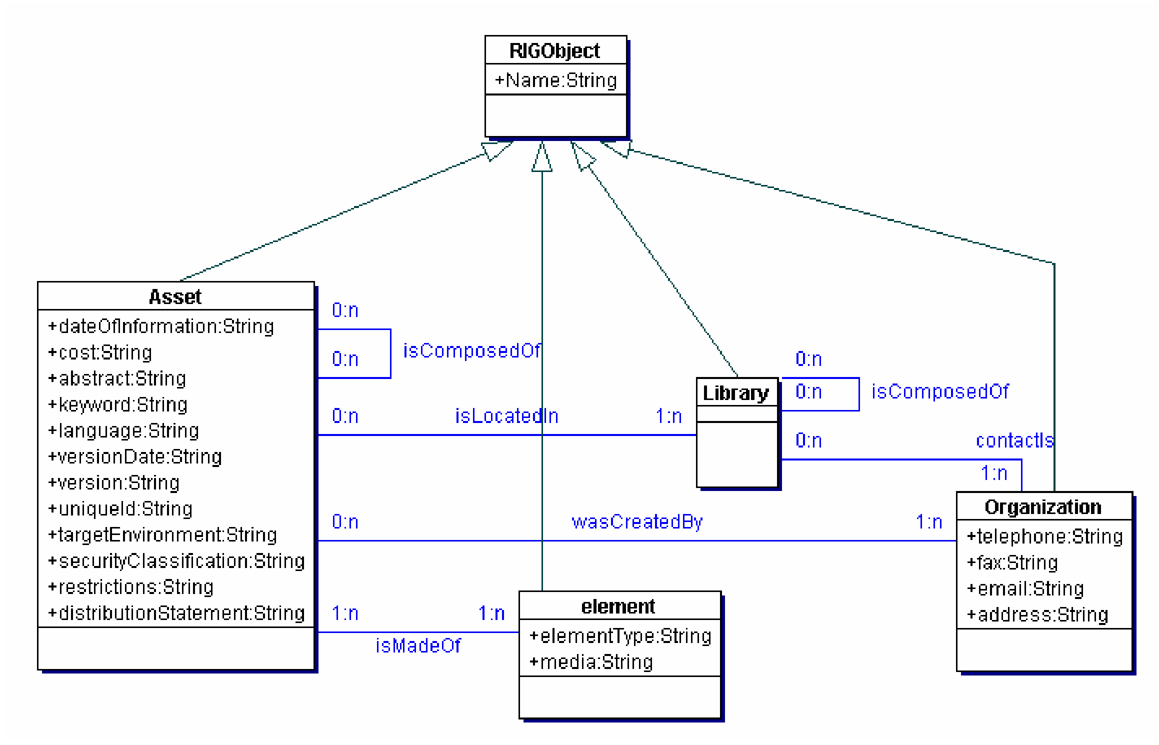


Figure 7 – Basic Interoperability Model

RIGObject is a base class that all other RIB classes inherit the “Name” attribute from. RIG stands for Reuse library Interoperability Group, an organization that collaborated with IEEE Software Engineering Standards committee to develop the Basic Interoperability Data Model[23]. RIGObject is extensible in a JBI context to a JBI Repository Object, a base object from which all content of the JBI repository inherit certain fundamental attributes (and perhaps methods as well).

The primary or most important class in the RIB data model is the Asset class. An asset in the RIB sense is a complete software package. The Asset class captures certain information about a persisted software package, such as a functional description (captured in the “abstract” attribute). Assets are constructed from elements, which can include source code, executables, documentation, etc. The attributes within the Element class identify the type of the element (e.g. source, executable, user’s manual etc.) and its storage media, which can be a compact disk, tape, a pointer to a file, or expressed as a universal resource identifier.

The primary or most important class in the JBI Repository data model is the Information Object. Currently, the Information Object is composed of type, metadata, and payload. A primary objective of experimentation with the RIB was to more precisely define the nature and composition of the Information Object.

Another objective of experimentation with the RIB was to capture the implicit use cases for the

RIB and to consider how extensions to those use cases would apply to the JBI Repository, if at all.

Approach

Initial installation consisted of unzipping an archive file which created a directory structure with subdirectories for Perl 5.6, MySQL 3.2.2 and Apache 1.3.6. Additionally, the administration interface requires a browser plug-in written in Java 1.1..

Upon installation, the first step to using the RIB was to gain familiarity with the BIDM. The element, asset, library, and organization tables are all interrelated and the RIB won't persist records unless all required data in each table has been entered. Once familiar with the model and with how constituent tables are related, the model and tables could be tailored to better meet project requirements.

Once the data model was tailored, entries were put into the library and organization tables. These tables are used to cluster data e.g. libraries cluster assets. After the library and organization tables have been populated, asset instances need to be entered into the assets table. Asset records contain the information that describes individual holdings. Assets cluster, or are composed of elements which are instantiated as files. Out of the box, all RIB data must be entered from the management interface one record at a time. Initially, assets should be entered, but the RIB forces the user to assign at least one element to each asset before allowing additional asset entries. The attributes of element were changed from required to optional, allowing all assets to be added to the RIB at once. After the RIB was populated with assets, elements were entered individually. The management interface was adequate for entering a limited number of records, but it was very labor intensive when entering a significant number of records. The evaluation project had 1500 records for inclusion into the RIB. A series of Perl scripts were developed that automatically loaded elements into the RIB, associated them with assets, and approved the elements for display. These scripts did not take advantage of the RIB API and a more general solution exploiting the full power of the API would be highly desirable.

As more knowledge was gained about the RIB, it became apparent that not all of its functionality was being utilized. Each asset and element has a keyword field that facilitates user searches. Users of the various instances of the RIB are generally familiar with the BIDM model and the content of various software catalogs. This familiarity provides a context for effective browsing, query, and retrieval of software, data, and documentation. Implementation of a RIB for the Rome Research Site requires the development and availability of a taxonomy (or context) that characterizes the holdings of the RRS RIB in a way presents a similar degree of context for users without requiring prior knowledge on their part. Coupled with the taxonomy, a naming convention should be developed that allows users to determine on inspection if particular assets are part of multiple assets.

Interoperability

One of the RIB’s strengths is interoperability. Two methods are provided: interoperability and synchronization. Interoperability involves *linking* one RIB instance with another. When two repositories are interoperating together, a user can access the assets of a remote RIB instance from his own. The assets are linked via URL. On the repository prototype test computer, the RIB instance was linked with 10 different repositories. RIB users on the test computer can now access over 100 different assets.

The other form of interoperability is synchronization. Synchronization involves physical replication of records contained in different RIB instances. A second RIB instance was created on a second test machine and was synchronized with the RIB instance on on the primary test machine. After the synchronization was complete, about 300 records were replicated between both RIB instances.

The only information required to accomplish either form of interoperability is the repository handle and the repository name[24]. With the required information, a user can interoperate with any other RIB instance via the web.

Initial Dataset

The initial data set loaded into our instance of the RIB consisted of over 1500 message transaction format (MTF) messages that were part of the Joint Warrior Interoperability Demonstration (JWID) 2000 scenarios. They were converted into XML using a conversion utility that was provided with the data. The conversion was accomplished using a Perl Script that transformed the MTF formatted messages into XML and then loaded the records into the RIB.

Making the RIB Operational

Initial steps have been taken to make the RIB operational in the AFRL/IF environment. When the RIB is initially loaded, the application is started only when the appropriate user logs in. When the user terminates his session, the RIB application is also terminated. Working with researchers at the University of Tennessee, our instance of the RIB was successfully integrated into a PKI enabled Apache web server as a cgi application. The next logical step is to get our instance of the RIB working using mod_perl instead of as a cgi application.

Current Repository Design

Over the course of our experimentation and evaluation, the current design of the JBI repository prototype has evolved as shown in the following figure

[24] The handle is a URL assigned to a RIB instance. The repository name is the name assigned to a RIB instance. For example www.software_catalog.rl.af.mil and “RRS Software Catalog”.

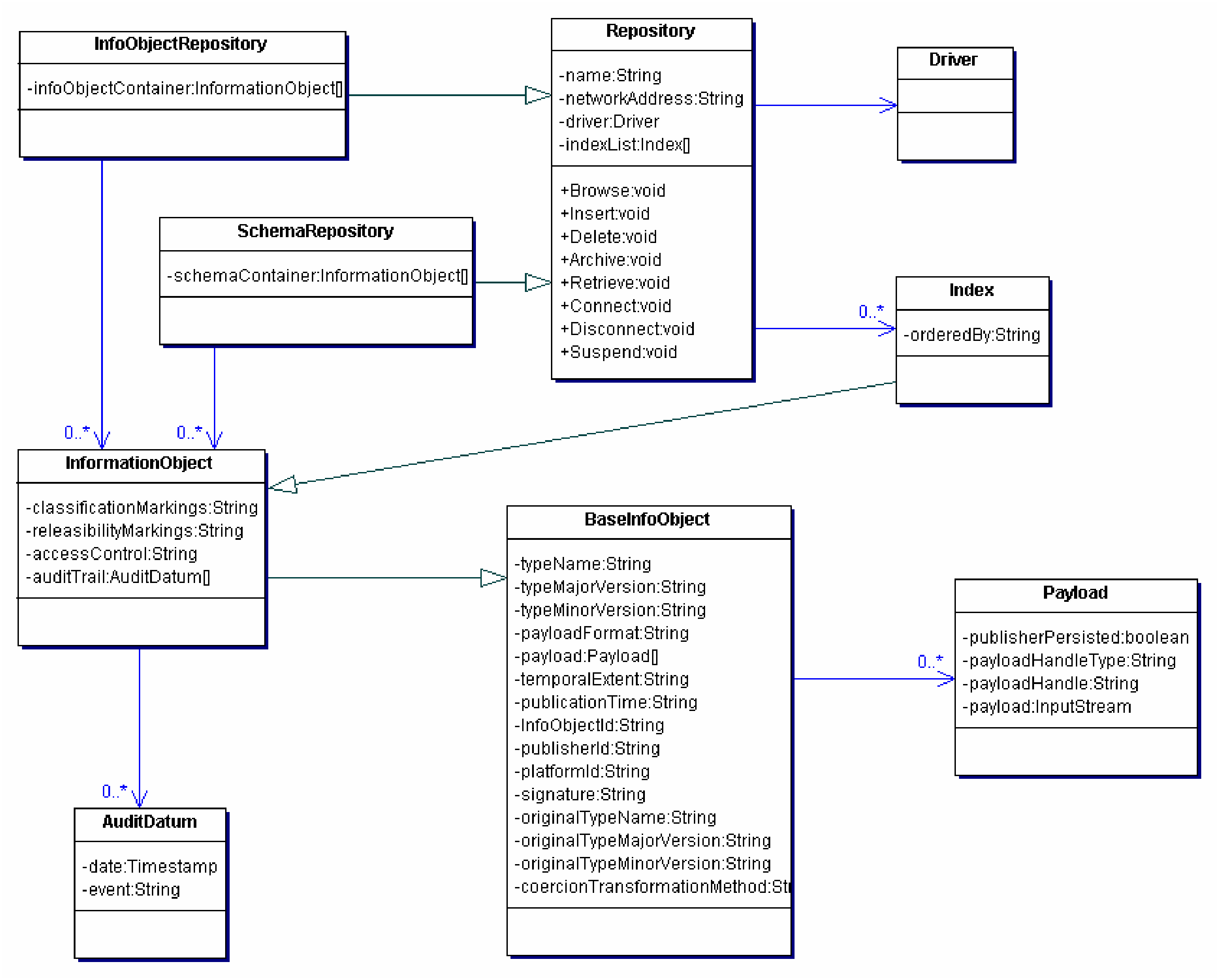


Figure 8 – Current Repository Object Model

There are currently nine classes in the repository design; BaseInfoObject, Payload, Index, Driver, InformationObject, AuditDatum, InfoObjectRepository, SchemaRepository, and Repository, of which two reflect actual repository implementations (InfoObjectRepository and SchemaRepository).

BaseInfoObject

The BaseInfoObject class is the fundamental building block of the object model. BaseInfoObject can be viewed as the point of departure for table layout (in relational parlance) in the repository proper. In the case of our design, the InformationObject class is a specialization of the BaseInfoObject class, inheriting all BaseInfoObject class attributes and operations, adding attributes for classification, releasibility, and access control markings as well as a container for audit data. The goal is to support all auditing requirements for system high accreditation by

Defense Intelligence Agency.

Strictly speaking, the JBI, and by extension the repository, do not process or care about an information object's payload, just the associated metadata instances required to support the query and publication/subscription core services. At a minimum therefore, the repository will have to contain copies of the metadata instances for all published information objects and pointers to the associated payload. In some cases, the payload will in fact reside in the JBI repository. More generally however, payload will be resident in external repositories.

Payload

The Payload class attribute publisherPersisted is true if the payload is physically located in an external repository, false if physically located within the JBI repository. The attribute payloadHandleType can be "handle"[25], "URI", "URL", "path", etc. The attribute payloadHandle is the string used to physically obtain external paylod. For example, if payloadHandleType = "path", payloadHandle could be "\\Lfs\projects\CYBERINFRA_SEMINAR\Anatomy of the Grid.pdf". Another example: if payloadHandleType = "URL", then payloadHandle could be http://www.rl.af.mil/programs/jbi/default.cfm. Or if payload is physically internal to the JBI repository, publisherPersisted would be false, payloadHandleType would be null, payloadHandle would be null, and the payload attribute would be non-empty.

Repository

There is an instance of the Repository class for every repository known to the JBI. An information object repository and a schema repository are structurally identical, but functionally distinct. Within an operational JBI, it is likely that the IOR will contain millions of information object instances, whereas the SR will contain thousands. Additionally, IOR content will be highly dynamic and its size will likely expand rapidly. SR content will be relatively static, it's size expanding relatively slowly.

The beauty of this approach is its syntactic and semantic simplicity. Relationships among information objects are readily represented as an information object of a specified type. Relationships among relationships, by extension are also readily represented as an information object of specified type. Hence, InfoObjectRepository and SchemaRepository are specializations of the Repository class.[26]

Index

The index class, while not implemented at this time, is also readily represented as an information

[25] See www.handle.net for details on the handle concept

[26] For an interestingly similar concept, see the Maya Report on the Universal Database by Peter Lucas and Jeff Senn, **Toward the Universal Database: U-forms and the VIA Repository,** Doc. No. MTR–02001, MAYA Design, Inc. The U-form concept is based on the e-forms concept first discussed by Michael Dertouzos (Former director of the Laboratory for Computer Science, and professor of computer science and electrical engineering at Massachusetts Institute of Technology) in his book *What Will Be: How the New World of Information Will Change Our Lives*

object of specified type. A repository instance may have zero or more indices, which can be passed as an argument in a query or a subscription. The purpose of the Index class is to represent some ordering of InformationObject contained within a Repository. Indices facilitate query, reporting, subscription and decision support, but generally indices exact a performance penalty during the update process[27] (i.e. publish). Well designed, carefully implemented indices minimize the overhead, poorly designed or implemented indices can ruin performance.

Client/Repository Interaction

It is important to stress at this point that only the InfoObjectRepository and SchemaRepository classes are exposed to clients only via platform core services through the common application program interface. Thus, clients interact with the JBI repository indirectly using the Browse, Insert, Delete, Archive, Retrieve, Connect, Disconnect, and Suspend operations the InfoObjectRepository and SchemaRepository classes inherit from the Repository class.

To quickly summarize functionality of the methods provided by the Repository class – Browse allows a client to list, or examine without action, the content of a particular repository. Insert allows new content to be added to a repository, while the appropriate indices are updated if required. Delete allows content to be permanently removed from a repository. Archive persists repository content to secondary or tertiary storage[28]. Retrieve copies content from the repository to a requesting client via query or subscription core services. Connect simply allows a client to hook up to a specific repository, again via the core services. Disconnect is the reverse of connect, and suspend allows a repository connection to remain in place, but inactive[29].

Conclusion

The JBI Repository Prototype work was able to accomplish the design and development of a prototype capability that provides repository services for a JBI. These services included the storage and retrieval of Information Objects and configuration information for the JBI itself. The objective in building the prototype was to obtain understanding of the requirements for a robust JBI repository, to examine alternative implementations in a laboratory setting, and this was accomplished.

Many potential technologies were investigated and employed. These technologies included Berkeley DB XML, Xindice from the Apache project, the open source XML:DB eXist, the open source relational database PostgreSQL, the Java compiler-compiler tool JavaCC, an XPath Parser Grammar usable by JavaCC, the XML-Java object binding project Castor and the RIB. All of these technologies contributed to the understanding of the functional capabilities required for the repository, insight into its design, and an understanding of the design space from which

[27] Inserting new objects into the repository at some point requires an insertion to the index or even a complete restructuring of the index.
[28] The JBI repository can be thought of primarily as an in-memory database. Archive forcibly writes content to disk. The mechanics for the appropriate criteria and thresholds are yet to be determined.
[29] A suspended connection requires that some degree of platform resources continue to be assigned until the connection becomes active again. In the event of a Disconnect, all platform resources are freed for other connections.

an operational implementation may be selected.

A repository design was implemented that is flexible, extensible, and scalable. The object model is straightforward and the mechanics of interaction with the repository are abstracted away from the clients. An object oriented approach to the design was selected precisely for this reason – a base design can be implemented and then extended without massive code redesign.

Future work will expand and extend the design shown in figure 8.